3.9 Working with Definitions

Throughout this course, we will study various mathematical objects that play key roles in computer science. As these objects become more complex, so too will our statements about them, to the point where if we try to write out everything using just basic set and arithmetic operations, our formulas won't fit on a single line! To avoid this problem, we create *definitions*, which we can use to express a long idea using a single term.[1]

[1] This is analogous to using local variables or helper functions in programming to express *part* of an overall value or computation.

In this section, we'll look at one extended example of defining our own predicates mathematically and in Python, and using them in our statements. Let us take some familiar terminology and make it precise using the languages of predicate logic and Python.

Definition. Let $n, d \in \mathbb{Z}$.[2] We say that d **divides** n,

[2] You may be used to defining divisibility for just the natural numbers, but it will be helpful to allow for negative numbers in our work.

or n **is divisible by** d, when there exists a $k \in \mathbb{Z}$ such that $n = dk$. In this case, we use the notation $d \mid n$ to represent "d divides n."

Note that just like the equals sign $=$ is a binary predicate, so too is $\mid$. For example, the statement $3 \mid 6$ is True, while the statement $4 \mid 10$ is False.[3]

[3] Students often confuse the divisibility predicate with the horizontal fraction bar. The former is a *predicate* that returns a boolean; the latter is a function that returns a number. So $4 \mid 10$ is $False$, while $\frac{10}{4}$ is 2.5.

This definition also permits $d = 0$, which may be a bit surprising! According to this definition, $0 \mid 0$, and for any non-zero $n \in \mathbb{Z}$, $0 \nmid n$.[4] In other

[4] Exercise: why are these two statements true?

words, when $d = 0$, $d \mid n$ *if and only if* $n = 0$.

Example. Let's express the statement "For every integer x, if x divides 10, then it also divides 100" in two ways: with the divisibility predicate $d \mid n$, and without it.

- **With the predicate**: this is a universal quantification over all possible integers, and contains a logical implication. So we can write

$$\forall x \in \mathbb{Z},\ x \mid 10 \Rightarrow x \mid 100.$$

- **Without the predicate**: the same structure is there, except we *unpack the definition* of divisibility, replacing every instance of $d \mid n$ with $\exists k \in \mathbb{Z},\ n = dk$.

  $$\forall x \in \mathbb{Z},\ \big(\exists k \in \mathbb{Z},\ 10 = kx\big) \Rightarrow \big(\exists k \in \mathbb{Z},\ 100 = kx\big).$$

  Note that each subformula in the parentheses has its own k variable, whose scope is limited by the parentheses.[5] However, even though

  [5] That is, the k in the hypothesis of the implication is different from the k in the conclusion: they can take on different values, though they can also take on the same value.

  this technically correct, it's often confusing for beginners. So instead, we'll tweak the variable names to emphasize their distinctness:

  $$\forall x \in \mathbb{Z},\ \big(\exists k_1 \in \mathbb{Z},\ 10 = k_1 x\big) \Rightarrow \big(\exists k_2 \in \mathbb{Z},\ 100 = k_2 x\big).$$

As you can see, using this new predicate makes our formula quite a bit more concise! But the usefulness of our definitions doesn't stop here: we can, of course, use our terms and predicates in further definitions.

Definition. Let $p \in \mathbb{Z}$. We say p is **prime** when it is greater than 1 and the only natural numbers that divide it are 1 and itself.

Example. Let's define a predicate $IsPrime(p)$ to express the statement that "p is a prime number," with and without using the divisibility predicate.

The first part of the definition, "greater than 1," is straightforward. The second part is a bit trickier, but a good insight is that we can enforce constraints on values through implication: *if a number d divides p, then $d = 1$ or $d = p$*. We can put these two ideas together to create a formula:

$$IsPrime(p) : p > 1 \wedge \big(\forall d \in \mathbb{N},\ d \mid p \Rightarrow d = 1 \vee d = p\big), \qquad \text{where } p \in \mathbb{Z}.$$

To express this idea without using divisibility predicate, we substitute in the definition of divisibility. The underline shows the changed part.

$$IsPrime(p) : p > 1 \wedge \big(\forall d \in \mathbb{N},\ \underline{(\exists k \in \mathbb{Z},\ p = kd)} \Rightarrow d = 1 \vee d = p\big), \quad \text{where } p \in \mathbb{Z}.$$

Expressing definitions in programs

As we just saw, in mathematics we can often express definitions as predicates, where an element of the domain (e.g., an integer) satisfies the predicate if it fits the definition. Because predicates are just functions, we can express these in programs as well. For example, let's consider the divisibility predicate $|$, where $d \mid n$ means $\exists k \in \mathbb{Z},\ n = kd$ (for $d, n \in \mathbb{Z}$). Here is the start of a function design in Python:

```
def divides(d: int, n: int) -> bool:
    """Return whether d divides n."""
```

While we can use the modulo operator `%` to implement this function (more on this later), we'll stick to remaining faithful to the mathematical definition as much as possible. Unfortunately, there is one challenge with translating the mathematical definition of divisibility precisely into a Python function. In mathematics we have no trouble at all representing an infinite set of numbers with the symbol $\mathbb{Z}$; but in a computer program, we cannot represent infinite sets in the same way. Instead, we'll use a *property* of divisibility to restrict the set of numbers to quantify over: when $n \neq 0$, every number that divides n must lie in the range $\{-|n|, -|n|+1, \ldots, |n|-1, |n|\}$.[6]

[6] We'll actually *prove* this property later on!

But the next question is, how do we represent the set $\{-|n|, -|n|+1, \ldots, |n|-1, |n|\}$ in Python, when the n is given as a parameter? We can use the `range` data type:[7]

[7] Remember the asymmetry here: the `start` argument is inclusive, but the `end` argument is exclusive.

```
possible_divisors = range(-abs(n), abs(n) + 1)
```

And then we can replace $\mathbb{Z}$ by this variable in the definition of divisibility to obtain $\exists k \in possible\_divisors,\ n = kd$. We can now translate this directly into Python code using what we learned earlier this chapter:

```
def divides(d: int, n: int) -> bool:
    """Return whether d divides n."""
```

```
    possible_divisors = range(- abs(n), abs(n) + 1)
    return any({n == k * d for k in possible_divisors})
```

Now let's turn our attention to the definition of $IsPrime$:

$$IsPrime(p) : p > 1 \wedge \big(\forall d \in \mathbb{N},\ d \mid p \Rightarrow d = 1 \vee d = p\big), \qquad \text{where } p \in \mathbb{Z}.$$

Here's a start for translating this definition into a Python function:

```
def is_prime(p: int) -> bool:
    """Return whether p is prime."""
```

Once again, we have a problem of an infinite set: $\forall d \in \mathbb{N}$. We can use the same property of divisibility as above and note that the possible natural numbers that are divisors of `p` are in the set $\{1, 2, \ldots, p\}$.[8] The quantified statement is a bit harder to translate because it

[8] This is simpler than the version above because $p \geq 1$.

contains an implication, so here we recall what we discussed in 3.3 Filtering Collections to use the `if` keyword in a comprehension to model implications. Here is our complete implementation of `is_prime`:

```
def is_prime(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(1, p + 1)
    return (
        p > 1 and
        all({d == 1 or d == p for d in possible_divisors if divides(d, p)})
    )
```

Notice that just like the mathematical definition, in Python our implementation of `is_prime` uses the `divides` function. This is a great example of how useful it can be to divide our work into functions that build on each other, rather than writing all of our code in a single function. As we learn about more complex domains in this course, we'll see this pattern repeat itself: definitions will build on top of one another, and you should expect that your functions will build on one another as well.

Divisibility and the remainder operation

You might have noticed that our definition of `divides`, though faithful to the mathematical definition, is not the same as how we've previously determined whether a number is divisible by 2 (i.e., is even).

```
def is_even(n: int) -> bool:
    """Return whether n is even."""
    return n % 2 == 0
```

In this case, we check whether `n` is divisible by 2 by checking whether the remainder when `n` is divided by 2 is 0 or not. It turns out that for *non-zero* $d \in Z$, checking remainders is equivalent to the original definition of divisibility:

$$\forall n, d \in \mathbb{Z},\ d \neq 0 \Rightarrow (d \mid n \Leftrightarrow n \ \% \ d = 0).$$

Note that when $d = 0$, the remainder $n \ \% \ d$ is undefined, and so we really do need the $d \neq 0$ condition in the above statement.

We can use this observation to write an alternate implementation of the `divides` function:

```
def divides2(d: int, n: int) -> bool:
    """Return whether d divides n."""
    if d == 0:
        # This is the original definition.
        possible_divisors = range(-abs(n), abs(n) + 1)
        return any({n == k * d for k in possible_divisors})
    else:
        # This is a new but equivalent check.
        return n % d == 0
```

You might also notice that the `d == 0` case is quite special: according to our definition of divisibility, when `d == 0` we know that `d` divides `n` if and only if `n == 0`:

$$\forall n, d \in \mathbb{Z},\ d = 0 \Rightarrow (d \mid n \Leftrightarrow n = 0)$$

We can use this to greatly simplify the if branch in our `divides2` function:

```
def divides3(d: int, n: int) -> bool:
    """Return whether d divides n."""
    if d == 0:
        # This is another new, equivalent check.
        return n == 0
    else:
        # This is a new but equivalent check.
        return n % d == 0
```

Our implementation in `divides3` meets the same function specification as the original `divides`, but has a much simpler implementation! It is also much more *efficient* than the original `divides`, meaning it performs fewer calculations (or computational "steps") and takes less time to compute its result. Intuitively, this is because the original `divides` function used the value `range(-abs(n), abs(n) + 1)` in a comprehension, and so the

number of expressions evaluated gets larger as `n` grows. This is not the case for `divides3`, which does not use a single `range` or comprehension in its body!

What this also means is that we can speed up our implementation of `is_prime` simply by calling `divides3` instead of `divides`:

```
def is_prime(p: int) -> bool:
    """Return whether p is prime."""
    possible_divisors = range(1, p + 1)
    return (
        p > 1 and
        all({d == 1 or d == p for d in possible_divisors if divides3(d, p)})
         # <-- Note the "divides3"
    )
```

This is a very powerful idea: we started out with one implementation of the divisibility predicate (`divides`), and then through some mathematical reasoning wrote a second implementation (`divides3`) that was logically equivalent to the first, but simpler and faster. But because `divides` and `divides3` were logically equivalent, we could safe replace `divides` with `divides3` in the implementation of `is_prime` to make it run faster, without worrying about introducing new errors!

The idea of *swapping out implementations* will come up again and again in this course. As the functions and programs you write grows larger, efficiency will be an important consideration for your code, and so it will be common to start with one function implementation and eventually replace it with another. We're only touching on these idea here with a relatively simple example, but we will talk formally about program efficiency in a later chapter.